Using Hierarchies for Optimizing Distributed Stream Queries

Sangeetha Seshadri, Vibhore Kumar and Brian F. Cooper
College of Computing, Georgia Institute of Technology
801 Atlantic Drive , Atlanta GA-30332
{sangeeta,vibhore,cooperb}@cc.gatech.edu

ABSTRACT

We consider the problem of query optimization in distributed data stream systems where multiple continuous queries may be executing simultaneously. In order to achieve the best performance, query planning (such as join ordering) must be considered in conjunction with deployment planning (e.g., assigning operators to physical nodes). In our scenario, the large number of network nodes, query operators, and opportunities for operator sharing between queries means that brute force and traditional techniques are too expensive. We propose two algorithms - the *Bottom-Up* algorithm and the *Top-Down* algorithm, which utilize hierarchical network partitions to provide scalable query optimization. We present analysis that establishes the bounds on the search-space and sub-optimality achieved by our algorithms. Finally, through simulations and experiments using a prototype deployed on Emulab [1] we demonstrate the effectiveness of our algorithms. The Top-Down algorithm, for instance, was able to achieve, on an average, solutions that were sub-optimal by only 10% while considering less than 1% of the search space.

1. INTRODUCTION

In many data stream systems, data is produced at multiple, geographically distributed sources. Examples include enterprise supply chain applications, scientific collaborations, and distributed network monitoring. It is often too expensive to stream all of the data to a centralized query processor, both because of the high communication costs, and the processing load at the central server. Instead, performing distributed processing of stream queries using techniques such as in-network processing [34, 23, 3] and filtering at the source [25] minimizes the communication overhead on the system and helps spread processing load, significantly improving performance. Then, we can think of a continual query as being "deployed" in the network, with data streams flowing between operators assigned to distributed physical nodes.

The typical approach used in distributed data stream systems [2, 29] is to construct a query plan (such as a relational algebra tree with a specified join ordering), deploy this plan into the network in some intelligent way, and then adapt the deployment at runtime to improve performance. This approach is shown in Figure 1. However, in this approach, several optimization opportunities may be lost: (1) the join order we choose may require intermediate results to be transported over a long distance, when an alternate join order would not; (2) the join order we choose may prevent us from reusing the results of an already deployed join from another query; (3) pushing selections to the source may similarly prevent us from reusing an already deployed sub-query; and so on. Of course, post-deployment adaptation can sometimes find these optimizations. However, if the initially deployed query plan is not very good, the adaptations may not find the optimizations, and may be less effective overall.

Figure 1: Typical Approaches (Plan, then deploy)

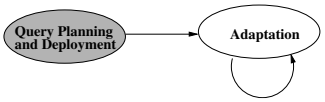

Figure 2: Our Approach. The shaded oval represents the phase where our algorithms can be used.

In order to take advantage of these optimizations, the query plan and the deployment must be considered simultaneously. In fact, as shown in Figure 3 significant ($> 50\%$) cost savings can be achieved by combining the planning and deployment phases. Our approach is summarized in Figure 2. As the figure shows, we can still use existing adaptive techniques once the query has been deployed. The basic idea of combining query planning and deployment has been proposed in [26, 28]; however, effective and scalable techniques for doing the planning and deployment together must be developed.

In this paper, we examine techniques for performing query planning in conjunction with deployment planning. A straightforward approach is to exhaustively enumerate all of the possible plans and deployments. This approach is used in traditional query planning and deployment in distributed

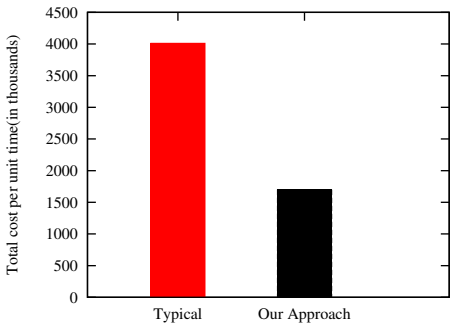

Figure 3: Comparison with typical approaches: The graph shows the total communication cost incurred by 100 queries over 5 stream sources each, on a 64-node network. The network topology was generated using the standard network topology generator GT-ITM. Our approach that considers query plans and deployments simultaneously reduces the cost by more than 50% as it was able to exploit optimization opportunities such as operator reuse.

databases [33] which typically considers only a few database nodes. Unfortunately, this approach is far too expensive for networks or query plans of even moderate size. For example, in the experiment shown in Figure 3 considering query plans and deployments simultaneously required us to examine 2.88×10^9 plans for a single query over 5 streams and that was only a 64 node network. (For the rest of the paper, we refer to each combination of query plan and operator placement simply as a 'plan'.) In order to make this problem tractable, we propose heuristics that trade some optimality to achieve a much smaller search space. In particular, we organize the physical nodes into a virtual hierarchy which along with "stream advertisements" is used to guide query planning and deployment and facilitate operator reuse. We introduce two algorithms. In the **Bottom-Up** algorithm, the query starts at the bottom of the hierarchy, and is propagated up the hierarchy, such that portions of the query are progressively planned and deployed. In the **Top-Down** algorithm, the query starts at the top of the hierarchy, and is recursively planned by partitioning the query and assigning sub-queries to progressively smaller portions of the network.

Although our algorithms may not find the optimal plan, we present analysis and experiments that show the sub-optimality is bounded. At the same time, our algorithms can reduce the search space by orders of magnitude compared to exhaustive search. For example, experimentally, the Top-Down algorithm was able to achieve, on average, solutions that were sub-optimal by only 10% while considering less than 1% of the search space.

This paper presents and analyzes approximation-based algorithms for the query optimization problem. In particular, we make the following contributions:

- We present a query optimization infrastructure that has two key components: a *hierarchical clustering* of network nodes (to help simplify the optimization problem), and *stream advertisements* (to enable operator reuse).
- We develop two algorithms, the **Bottom-Up** and **Top-Down** algorithms, for finding an efficient execution plan. These algorithms utilize the hierarchy and advertisements of our optimization infrastructure.
- We provide well-characterized bounds on the search space and sub-optimality offered by each algorithm. We show analytically that the Top-Down algorithm is more effective than the Bottom-Up algorithm at bounding the sub-optimality of the deployment. While the two algorithms have the same worst-case search space size, the Bottom-Up offers a smaller search space on average, because it prunes plans more aggressively.
- We present the results of an experimental study evaluating the effectiveness of our techniques, conducted using simulations and a prototype deployed on Emulab.

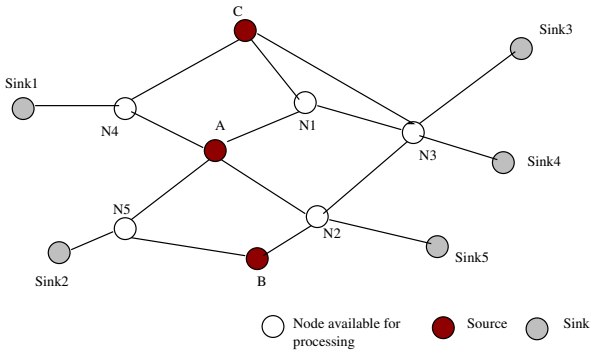

Figure 4: An example network N

1.1 An Illustrative Example

Let us examine the optimization opportunities that may be available in a distributed data stream system where multiple continuous queries may be executing simultaneously. Consider a distributed stream system operating over a network N shown in Figure 4. Let A, B and C represent sources of data-streams and nodes $N1-N5$ be available for in-network processing. Each line in the diagram represents a physical network link. Let us also assume that we can estimate the expected data-rates of the stream sources and the selectivities of their various attributes, perhaps gathered from historical observations of the stream-data or measured by special purpose nodes deployed specifically to gather data statistics.

Imagine that the following query Q1 arrives at sink $Sink4$.

Q1: `SELECT * FROM A⋈B⋈C WHERE` $F_A \wedge F_B \wedge F_C$

($F_A \wedge F_B \wedge F_C$ is a filter predicate over the streams.)

1. **Network-aware join ordering:** Based purely on the size of intermediate results, we may normally choose the join order (A⋈B)⋈C. Then we would deploy the join A⋈B at node $N2$, and the join with stream C at node $N3$. However, node $N2$ may be overloaded, or the link A$\rightarrow N2$ may be congested. In this case, the network conditions dictate that a more efficient join ordering is (A⋈C)⋈B, with A⋈C deployed at $N1$, and the join with B at $N3$.

Now, consider situations where we may be able to reuse an

already deployed operator. This will reduce network usage (since the base data only needs to be streamed once) and processing (since the join only needs to be computed once). Imagine that query Q2 has already been deployed:

Q2: SELECT * FROM A⋈C WHERE $F_A \wedge F_C$

with the join A⋈C deployed at $N1$. Assume that the sink for the query Q2 is located at node $Sink3$.

2. **Operator Reuse:** Although the optimal operator ordering in terms of the size of intermediate results for query Q1 may be (A⋈B)⋈C, in order to reuse the already deployed operator A⋈C, we must pick the alternate join ordering (A⋈C)⋈B. In contrast, if the sinks for the two queries are far apart (say, at opposite ends of the network), we may decide not to reuse Q2's join; instead, we would duplicate the A⋈C operator at different network nodes, or use a different join-ordering. Thus, having knowledge of already deployed queries influences our query planning.
3. **Delayed Filtering:** Normally, we would push the filter predicates F_A, F_B and F_C towards the sources. However, doing so may prevent us from re-using Q2's deployed A⋈C, if the predicates for Q2 are different than Q1. In this case, we may decide to delay filtering, until after the join; although this violates the traditional rule of thumb for "pushing down selections," because it enables reuse of the join, it may be more efficient overall.

These examples show that the network conditions and already deployed operators must often be considered when choosing a query plan and deployment in order to achieve the highest performance.

1.2 Paper overview

The remainder of this paper is organized as follows - a formal description of the query optimization problem is presented in Section 2. In Section 3 we present our algorithms and rigorously analyze their effectiveness. An experimental evaluation of the proposed solutions is presented in Section 4. We discuss related work in Section 5 and finally conclude in Section 6 with a discussion of possible future directions.

2. PROBLEM DEFINITION

We consider the query optimization problem for multiple continuous queries that may be executing simultaneously in a distributed data stream system. Our problem definition addresses the continual query equivalent of 'select-project-join' queries that involve simple selection, projection and join operations on one or more data streams. The focus of this paper is on join-ordering and selections. We leave queries involving aggregations and unions to future work. Joins over stream data may be performed using a variety of techniques such as windowed joins and symmetric hash joins. However, our system model is based on a generalized notion of joins and is not tied to any particular technique. The underlying system, described formally shortly, is essentially a collection of network nodes and the links between them. We assume that potentially, *any* operator can be deployed at *any* node in the system. Given a query, there could possibly be multiple execution plans that the system could follow to produce results. We assume that all such plans produce equivalent results.

2.1 System Definition

We now formally describe the components of our distributed data stream system. Let $N(V_n, E_n)$ represent a physical network of nodes where vertices V_n represent the set of actual physical nodes and the network connections between the nodes are represented by the set of edges E_n.

Let Q represent a single continuous query and let $P^Q = \{p_1^Q, \ldots, p_m^Q\}$ represent the set of all relational algebra query trees (e.g. operator orderings) for query Q. Each query tree p_j^Q can be represented as a directed acyclic graph $G(V_j^Q, E_j^Q)$ where each element v_{jk}^Q of set V_j^Q represents either a source, operator or sink. While sources and the sink have a static association with a vertex in graph N, operator vertices can be dynamically associated with any vertex in the graph N. For each operator vertex v_u^Q, we can estimate the selectivity by measuring the ratio of outgoing to incoming data. For example, filters and selective joins reduce dataflow, while other joins may increase the dataflow.

The *deployment* of a query tree p_j^Q over the network N is defined as a mapping $M(p_j^Q, N)$ that assigns each vertex $v_{jk}^Q \in V_j^Q$ to a network node $v_{nk} \in V_n$. Thus, M implies a corresponding mapping of edges in G to edges in N. That is, each edge e_{juw}^Q between operators v_{ju}^Q and v_{jw}^Q is mapped to the network edges along the lowest cost path between the network nodes to which v_{ju}^Q and v_{jw}^Q are assigned. Thus, for each $v_{jk}^Q \in V_j^Q$ we have $M(v_{jk}^Q) \in V_n$ and similarly for each $e_{juw}^Q \in E_j^Q$ we have $M(e_{juw}^Q) \subseteq E_n$.

2.2 Optimization Criteria

In a distributed data-stream system where communication and processing costs are high and incurred continuously, an optimal query execution plan should ideally try to achieve multiple objectives - a minimum response-time while incurring a minimum communication and processing cost per unit time. However, these objectives may be conflicting, since it is possible that lower delay paths have higher communication cost, or it may be the case that paths that incur low communication cost can cause a processing overload at some intervening network node. To optimize across such conflicting objectives we choose to use the approach suggested in [17], where the optimization criteria is some 'application dependent' cost function expressed in terms of objectives, which in our case are the response time, communication cost and the processing cost.

In many scenarios, such as mission critical real-time applications, the optimization criteria may degenerate to the problem of minimizing the response-time and in other scenarios, like power constrained sensor networks, it may take up a formulation that aims to minimize the communication and the processing cost. The query optimization algorithms presented later in this paper are capable of incorporating any 'application dependent' cost function to find an efficient query execution plan. Henceforth, we will use $Cost(M(p_i^Q, N))$ to denote the 'application dependent' cost

incurred by a deployment M of a query tree p_i^Q of a query Q over the network N. In particular, the sub-optimality analysis and the experiments reported in this paper use a cost formulation that tries to minimize the communication cost incurred per unit time by the deployed query plan.

2.3 Optimization Problem

We now define the query-optimization problem for queries to be deployed on a distributed data stream system.

Query-Optimization Problem: Given a query Q to be deployed over a network N, and a (possibly empty) set of existing query deployments $D = \{D_1,\ldots,D_n\}$, find a query tree $\{p_i^Q\}$ and a deployment $M(p_i^Q, N)$ for Q such that $Cost(M(p_i^Q, N))$ is minimum over all possible query trees and deployments of such trees.

The problem is to find an optimal deployment for a query, given a set of base stream sources and a set of query deployments already executing on the system, by taking into consideration operator reuse. This problem is similar to that of computing optimal query trees in database systems while utilizing existing materialized views [14], but takes into consideration the additional dimension of network costs. Note that it may be possible to modify existing deployments to get a better solution. However, such modifications require us to consider the cost of re-configurations as well. We leave such possibilities for the future.

3. QUERY OPTIMIZATION ALGORITHMS

In order to choose an optimal execution plan, traditional query optimizers typically use an exhaustive search of the solution space, estimating the cost of each plan using pre-computed statistics. Lemma 1 shows the size of the exhaustive search space for the query optimization problem in distributed data stream systems.

LEMMA 1. *Let $\mathcal{Q}$ be a query over K (> 1) sources to be deployed on a network with N nodes. Then the size of the solution space of an exhaustive search is given by:*

$$\mathcal{O}_{\texttt{exhaustive}} = \left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (N)^{(K-1)}$$

PROOF. We are given a network with N nodes, and a query $\mathcal{Q}$ over K streams $\langle \mathtt{S}_1, \mathtt{S}_2, \ldots \mathtt{S}_K \rangle$. The search space is given by all plans (permutations of join-orders) and all possible placements of each plan. The number of query re-writings i.e. an enumeration of both linear and bushy joins of K streams is given by:

$$\binom{K}{2} + \binom{K-1}{2} + \ldots + \binom{2}{2} = \frac{K! \times (K-1)!}{2^{K-1}}$$

The number of network placements of the joins in a query with K streams in a network of size N is given by $N^{(K-1)}$. Thus, the exhaustive search space $\mathcal{O}_{exhaustive}$ given by:

$$\mathcal{O}_{\texttt{exhaustive}} = \left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (N)^{(K-1)}$$

□

As shown in the Lemma 1, the search space increases exponentially with an increase in the query size. Certainly, in a system with thousands of nodes such an exhaustive search would be infeasible.

We have developed two approximation-based algorithms for query optimization in a distributed data stream system. Given a query and a set of sources, these algorithms find an efficient execution plan by simultaneously taking into consideration operator ordering, reuse and network placement. Both algorithms depend on an underlying infrastructure that provides: 1. a hierarchical clustering of network nodes based on our cost function and 2. *stream advertisements* that allow the query planner to determine where stream data, both base data and data derived from existing operators, are available.

- The **Bottom-Up algorithm** begins by deploying a query in the cluster containing the query sink. It then works its way up the cluster hierarchy, progressively decomposing the query into pieces: one that can be answered using data streams in the local cluster, and one which must be passed further up the hierarchy in order to discover remote streams.
- The **Top-Down algorithm** starts at the top level of the hierarchy, and performs an exhaustive search using the whole query to find an execution plan. However, the exhaustive search only examines a set of "representative nodes" that approximate the properties of many other nodes. Each of these representative nodes is assigned a piece of the query, and similarly uses exhaustive search to deploy that piece in its own cluster. The algorithm works its way down the hierarchy, progressively decomposing the query into pieces that are assigned to finer-grained clusters.

Both algorithms avoid the cost of exhaustive search by limiting the search space to subsets of the network. Moreover, the algorithms operate in a distributed manner, using localized network and data statistics, avoiding the high cost of a centralized planner which must track the properties of the entire network.

We now describe our algorithms in detail. First, we describe the infrastructure that support the algorithms. Then, we discuss each algorithm, and analyze bounds on the search space and plan sub-optimality. Finally, we compare and contrast the two algorithms.

3.1 Optimization infrastructure

In this section we describe the key components of our optimization infrastructure - *hierarchical network partitions* that guide our planning heuristics and *stream advertisements* that facilitate operator reuse.

We can tune the hierarchy to trade off between search space size and sub-optimality by adjusting the max_{cs} parameter, which bounds the maximum size of each network partition. This tradeoff is complex, and is analyzed in detail in our discussion of the Bottom-Up (Section 3.2) and Top-Down (Section 3.3) algorithms.

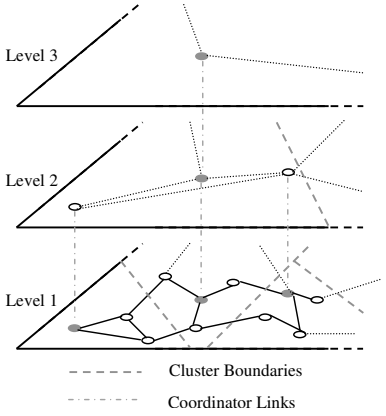

Figure 5: Hierarchical network clusters

3.1.1 Hierarchical Network Clusters

We organize physical network nodes into a virtual clustering hierarchy, by clustering nodes based on our optimization criteria. For example, if the metric is response-time, we cluster based on inter-node delays. If the metric is communication costs, we cluster based on link costs which represents the cost of transmitting a unit amount of data across the link connecting the two nodes. We refer to this clustering parameter as *inter-node traversal cost*. Nodes that are close to each other in the sense of this clustering parameter are allocated to the same cluster. However, we impose an upper bound on the cluster size, specified as the max_{cs} parameter.

Clusters are formed into a hierarchy. At the lowest level, i.e. *Level 1*, the physical nodes are organized into clusters, in accordance with clustering parameter max_{cs}. Each node within a cluster is aware of the inter-node traversal cost between every pair of nodes in the cluster. A single node from each cluster is then selected as the *coordinator* node for that cluster and promoted to the next level, *Level 2*. Nodes in *Level 2* are again clustered according to average inter-node traversal cost, with the cluster formation being governed by the parameter max_{cs}. This process of clustering and coordinator selection continues until *Level N* where we have just a single cluster. An example hierarchy is shown in Figure 5. As a result of our clustering approach we can determine the upper bounds on the cost approximation at each level, which is described in the following theorem.

THEOREM 1. *Let d_i be the maximum intra-cluster traversal cost at level i in the network hierarchy and $c_{act}(v_{nj}, v_{nk})$ be the actual traversal cost between the network nodes v_{nj} and v_{nk}. Then the estimated cost $c^l_{est}(v_{nj}, v_{nk})$ between the same network nodes at any level l is related to the actual cost as follows:*

$$c_{act}(v_{nj}, v_{nk}) \leq c^l_{est}(v_{nj}, v_{nk}) + \sum_{i=1}^{i<l} 2d_i$$

PROOF. At a particular level l the cost of traversal between nodes v_{nj} and v_{nk} is given by the inter-node traversal cost between the nodes representing them at that level. However, each node will be resolved to some node in the underlying cluster at level $l-1$. Inter-node traversal costs at this level are bounded by the value d_{l-1}. Therefore, nodes at level $l-1$ will be atmost d_{l-1} distance away from the node representing them at level l. Thus the inter-node traversal costs between nodes v_{nj} and v_{nk} at level $l-1$ is given by

$$c^{l-1}_{est}(v_{nj}, v_{nk}) \leq c^l_{est}(v_{nj}, v_{nk}) + 2d_{l-1}$$

. Similarly,

$$c^{l-2}_{est}(v_{nj}, v_{nk}) \leq c^{l-1}_{est}(v_{nj}, v_{nk}) + 2d_{l-2}$$

.

$$\Rightarrow c^{l-2}_{est}(v_{nj}, v_{nk}) \leq c^l_{est}(v_{nj}, v_{nk}) + \sum_{i=l-2}^{i<l} 2d_i$$

. This process continues down the hierarchy. At level 1, the estimated cost is the same as the actual traversal cost. Therefore the estimated traversal cost at level l is at most $\sum_{i=1}^{i<l} 2d_i$ less than the actual cost. □

The hierarchical organization is created and maintained using the following algorithm. When a node joins the infrastructure, it contacts an existing node that forwards the join request to its coordinator. The request is propagated up the hierarchy and the top level coordinator then assigns the new node to the top level node that is nearest to it. This top level node passes the request down to its child that is closest to the new node. This child repeats the process, which continues until the node is assigned to a bottom level cluster. Note that similar organization strategies appear in other domains such as hierarchies for internet routing [24], for data aggregation in sensor networks [8] and other related applications. However, to the best of our knowledge we are the first to use such hierarchical approximations and clustering techniques for distributed continual query optimization.

The virtual hierarchy is robust enough to adapt as necessary. It can handle both node joins and departures at runtime. Failure of coordinator nodes can be handled by maintaining active back-ups of the coordinator node within each cluster. However, the issue of fault tolerance is beyond the scope of this paper [22]. Note that, given a network, multiple clustering hierarchies can be created simultaneously with different values of the max_{cs} parameter.

3.1.2 Stream Advertisements

Stream Advertisements are used by nodes in the network to advertise the stream sources available at that node. A node may advertise two kinds of stream sources - *base stream sources* and *derived stream sources*. We observe that each sink and deployed operator is a new stream source for the data computed by its underlying query or sub-query. We refer to these stream sources as derived stream sources and the original stream sources as base stream sources. As a result of the advertisement of derived stream sources, nodes are now aware of operators that are readily available at multiple locations in the network and can be reused with no additional cost involved for transporting input data. The stream advertisements are aggregated by the coordinator nodes and propagated up the hierarchy. Thus the coordinator node at each level is aware of all the stream sources available in its underlying cluster. Advertisements of derived stream

sources are key to operator reuse in our algorithms. The advertisements are one-time messages exchanged only at the initial time of operator creation and deployment.

3.2 The Bottom-Up Algorithm

We now describe the *Bottom-Up* algorithm which propagates queries up the hierarchy, progressively constructing complete query execution plans as and when the required information is available. The algorithm is formally described in Figure 6.

Queries are registered at their sink. When a new query $\mathcal{Q}$ over base stream sources $\langle S_1, S_2, \ldots S_i \rangle$ arrives at a sink at *Level 1*, the sink informs its coordinator at *Level 2*. The coordinator rewrites the query $\mathcal{Q}$ as $\mathcal{Q}'$ with respect to two views - $V^{\mathcal{Q}}_{local}$ and $V^{\mathcal{Q}}_{remote}$ where $V^{\mathcal{Q}}_{local}$ is composed of base and derived sources available locally within the cluster and $V^{\mathcal{Q}}_{remote}$ is composed of base sources not available locally (steps 3-8 in Figure 6). Thus $\mathcal{Q}' \leftarrow V^{\mathcal{Q}}_{local} \bowtie V^{\mathcal{Q}}_{remote}$. The coordinator deploys $V^{\mathcal{Q}}_{local}$ within the current cluster (step 10), and then advertises $V^{\mathcal{Q}}_{local}$ as a derived stream at the next level. The above rewriting causes any joins between local streams to be deployed within the current cluster, leaving the joins of local streams with remote streams or joins between remote streams to be deployed further up in the hierarchy. It may be noted that $V^{\mathcal{Q}}_{local}$ may project a few columns that are not projected in $\mathcal{Q}$ to facilitate the join with the remote view. Furthermore, the remote view is composed only of base sources since each node is only aware of derived sources available within its own cluster and not aware of those in remote clusters.

The coordinator then requests $\mathcal{Q}'$ from its next level coordinator, who similarly decomposes the query and repeats the process (step 16). This process continues up the hierarchy, with the query $\mathcal{Q}'$ progressively decomposed into locally available views and remote views and the re-written query being requested from the current cluster's coordinator. Meanwhile, the coordinator performs an exhaustive search, within its underlying cluster, to determine an optimal execution plan for $V^{\mathcal{Q}}_{local}$. By limiting exhaustive searches to only sub-queries that can be composed within a single partition and only to the current coordinator's partition, the Bottom-Up algorithm is able to bound the search-space of possible execution plans and their deployments.

By considering all possible constructions of $V^{\mathcal{Q}}_{local}$ that utilize derived sources, the coordinator at each level takes into account reuse of operators already existing in its underlying cluster. When using a derived stream source, communication costs for transporting input data to the node that is the source of the derived stream, and processing costs for computing the result of the operator are incurred only once. Note that if it is cheaper to duplicate operators rather than reuse existing ones, the coordinator will do so.

3.2.1 Bounding Search Space with the Bottom-Up Algorithm

In a network N that is organized into a clustering hierarchy, for a query $\mathcal{Q}$ over K (> 1) sources the search space depends on the clustering parameter max_{cs} and the resulting height

Algorithm: Bottom-Up
Input: Query $\mathcal{Q}$ over base-sources $S[1..m]$ arrives at *this* node
Output: Deployed query plan

```
bottomUp (Query Q, Sources S[1..m])
1.  localSources[] := {}, remoteSources[] := {};
2.  for(i := 1; i < m; i++)
3.    if(S[i].isVisibleAt(N))
4.      localSources.addSource(S[i]);
5.    else
6.      remoteSources.addSource(S[i]);
7.    end;
8.  end;
9.  localQuery := createSubQuery(Q, localSources);
   //Perform exhaustive search for sub-query
   //within the local cluster
10. localPlan := this.deployQuery(localQuery);
11. if(remoteSources = null)
12.  return localPlan;
13. else
14.  remoteSources.addSource(localPlan.sink);
15.  Q' := createSubQuery(Q, remoteSources);
16.  return this.coordinator->bottomUp(Q', remoteSources)
    + localPlan;
17. end;
```

Figure 6: Bottom-Up Algorithm

$h (\approx log_{max_{cs}} N)$ of the hierarchy. We define the following:

$$\beta = h(\frac{max_{cs}}{N})^{K-1} \quad (1)$$

In Theorem 2 we prove that β represents the upper bound on the ratio of the search space of the Bottom-Up algorithm to that of the exhaustive search. Note that as the ratio $\frac{max_{cs}}{N}$ decreases linearly, β decreases exponentially. When $max_{cs} << N$, β is orders of magnitude less than 1 and thus, the Bottom-Up algorithm is orders of magnitude cheaper than exhaustive search. For example, for a query over 4 streams on a network with 1000 nodes, with a max_{cs} value of 100, $\beta \approx 0.0015$.

THEOREM 2. *Let $\mathcal{Q}$ be a query over K (> 1) sources to be deployed on a network with N nodes. Let the clustering parameter used to organize the network into a hierarchical cluster be max_{cs} and let the height of such a hierarchical cluster be h. Let $\mathcal{O}_{bottom-up}$ represent the solution space for the bottom-up algorithm. Then,*

$$\mathcal{O}_{bottom-up} \leq \beta \mathcal{O}_{exhaustive}$$

PROOF. We are given a network with N nodes, and a query $\mathcal{Q}$ over K streams $\langle S_1, S_2, \ldots S_K \rangle$. Let σ_i represent the number of streams, for query Q, requested by a node at level $i-1$ and available within the partition of a single coordinator at level i. Also, $\sigma_1 + \ldots + \sigma_h = K$. Let α_i represent the actual number of streams to be considered at level i. At the level where $V^{\mathcal{Q}_i}_{remote} = \phi$, $\alpha_i = \sigma_i$. At all other levels $\alpha_i = \sigma_i + 1$ to take into consideration the presence of the remote stream $V^{\mathcal{Q}_i}_{remote}$. Thus, $\alpha_1 + \ldots + \alpha_h \leq K + h$. At any level the search space is given by all plans (permutations of join-orders) and all possible placements of each plan. The number of query re-writings at level i i.e. an enumeration of both linear and bushy joins of α_i streams is given by:

$$\binom{\alpha_i}{2} + \binom{\alpha_i - 1}{2} + \ldots + \binom{2}{2} = \frac{\alpha_i! \times (\alpha_i - 1)!}{2^{\alpha_i - 1}} \quad (2)$$

The number of network placements of the joins in a query with α_i streams in a cluster of size $\max_{cs}$ is given by $(max_{cs})^{(\alpha_i - 1)}$. Thus the search space $\mathcal{O}_i$ at level `i` is given by:

$$\mathcal{O}_i \leq \left(\frac{\alpha_i! \times (\alpha_i - 1)!}{2^{\alpha_i - 1}} \right) \times (max_{cs})^{(\alpha_i - 1)}$$

Thus the total search space in the Bottom-Up algorithm, $\mathcal{O}_{bottom-up}$ for a query `Q` is:

$$\mathcal{O}_{\texttt{bottom-up}} \leq \sum_{i=1}^{i \leq h} \mathcal{O}_i \tag{3}$$

Since $\forall$ i, $\alpha_i \leq K$, and not all $\alpha_i = K$ (since the query is totally composed of only K streams and streams found at each level are different), we have

$$\mathcal{O}_{bottom-up} \leq \sum_{i=1}^{i \leq h} \left(\frac{K! \times (K-1)!}{2^{K-1}} \right) \times (max_{cs})^{(K-1)}$$

$$\Rightarrow \mathcal{O}_{bottom-up} \leq \left(\frac{K! \times (K-1)!}{2^{K-1}} \right) \times (max_{cs})^{(K-1)} \times (h) \tag{4}$$

Thus, from Lemma 1 and Equation 4 we have:

$$\mathcal{O}_{bottom-up} \leq \beta \mathcal{O}_{exhaustive}$$

□

3.2.2 Sub-Optimality in the Bottom-Up Algorithm

The Bottom-Up algorithm partitions queries into locally and remotely available views as the result of which all local sources are now represented as a single source deployed at the coordinator. This results in a pruning of the plan search space since only join orderings between streams available within a single cluster are considered. While the Bottom-Up algorithm can find optimal join orderings among local sources, the resulting overall execution plan may be sub-optimal. As an example, consider a high volume stream S_r that is in a remote cluster, and which we want to join with two low volume, local streams S_1 and S_2. An overall optimal plan might be to perform a selective join between S_r and S_1 in the remote cluster, and then stream the resulting (low-volume) intermediate results to the local cluster for joining with S_2. The Bottom-Up algorithm will not consider this plan. However, note that the Bottom-Up algorithm may instead stream the results of $S_1 \bowtie S_2$ to the remote cluster for joining with S_r.

In the worst case the resulting deployment may be arbitrarily bad making it impossible to bound the sub-optimality of the algorithm. However, note that the situations under which this algorithm performs badly can be well characterized: the algorithm performs badly when streams available remotely have significantly higher data rates than those available close to the sink. Therefore, it is possible to identify these scenarios a priori through static analysis of stream rates and selectivities and use the Top-Down algorithm in those cases.

We show in Theorem 3, that the sub-optimality of the plan chosen by the Bottom-Up algorithm is bounded with respect to the most optimal deployment of the same join-ordering.

Algorithm: Top-Down
Input: Query $\mathcal{Q}$ over base-sources $S[1..m]$ arrives at *this* node
Output: Deployed query plan

```
topDown (Query Q, Sources S[1..m])
 1. return this.topLevelCoordinator->recurse(Q, S);
recurse (Query Q, Sources S[1...m])
 1.   plan := this.deployQuery(Q, S);
 2.   n := |this.clusterSize|;
 3.   if(n = 0)
 4.    return plan;
 5.   else
 6.    {subQ[1..n],subS[1..n][]}:=distributeToClusters(plan);
 7.    for(i := 1; i < n; i++)
 8.     plan += this.cluster[i]->recurse(subQ[i], subS[i]);
 9.    end;
 10.   return plan;
 12.  end;
```

Figure 7: Top-Down Algorithm

This proves that the Bottom-Up algorithm can offer better bounds than a random placement of the same query tree. Thus, the Bottom-Up algorithm is ideal in situations where the network placement of operators is a more dominant factor than join-ordering.

THEOREM 3. *A query $\mathcal{Q}$ deployed as query tree $\mathcal{P}$ over a network N using the Bottom-Up algorithm is no more than*

$$\sum_{l=1}^{l=h-1} \left(\sum_{j=1}^{j=2\alpha_l - 1} \left(\sum_{i=1}^{i=l-1} 2d_i \right) \times s_j \right)$$

sub-optimal compared to the optimal deployment of query tree $\mathcal{P}$ over the same network N, where h is the number of levels in the hierarchical organization of N, α_l represents the number of query sources found at a certain level l in the network hierarchy, d_i is the maximum intra-cluster delay at level i and s_j represents the stream rate of the j^{th} edge deployed at that level.

PROOF. Assume that the query $\mathcal{Q}$ has K sources, and the network N is organized into a clustering hierarchy of height h with $\{d_1, d_2, ...d_h\}$ as the maximum intra-cluster traversal cost at the corresponding level. Let α_l represent the number of sources found at a certain level l in the network hierarchy. The number of edges of plan $\mathcal{P}$ therefore deployed at any level l is equal to $2\alpha_l$ - 1. If s_j be the stream-rate corresponding to the j^{th} edge deployed at level l, the corresponding error is given by: $\sum_{j=1}^{j=2\alpha_l - 1} (\sum_{i=1}^{i=l-1} 2d_i) \times s_j$ (refer Theorem 1). The total error, i.e. the sum of errors across all levels is

$$\sum_{l=1}^{l=h} \left(\sum_{j=1}^{j=2\alpha_l - 1} \left(\sum_{i=1}^{i=l-1} 2d_i \right) \times s_j \right)$$

□

3.3 The Top-Down Algorithm

The *Top-Down* algorithm (Figure 7) bounds sub-optimality by making deployment decisions using bounded approximations of the underlying network; specifically, each coordinator's estimate of the distance between its cluster and other clusters. The algorithm works as follows: The query $\mathcal{Q}$ is

submitted as input to the top level (say level t) coordinator (step 1 of sub-routine **topDown** in Figure 7). The coordinator exhaustively constructs the possible query trees for the query, and then for each such tree constructs a set of all possible node assignments within its current cluster. The cost for each assignment is calculated and the assignment with least cost is chosen. An assignment of operators to nodes partitions the query into a number of views, each allocated to a single node at level t (step 1 of sub-routine **recurse**). Each node is then responsible for instantiating such a view using sources (base or derived) available within its underlying cluster. The allocated views act as the queries that are again deployed in a similar manner at level $t-1$, with all possible assignments within the cluster being evaluated exhaustively and the one with the least cost being chosen (steps 6-9). This process continues until level 1, which is the level at which all the physical nodes reside, and operators are assigned to actual physical nodes (step 3-4).

The hierarchical structure and limited size of partitions minimizes the expense of the exhaustive mapping performed by each coordinator. Since each level has fewer nodes and operators are progressively partitioned and assigned to different cluster coordinators, the search space is still much smaller compared to a global exhaustive search.

Whenever a coordinator is exhaustively mapping a portion of the query, it considers both base and derived streams that are locally available. As a result, operator reuse is considered automatically in the planning process. In particular, if the coordinator calculates that reuse would result in the best plan, derived streams are used; otherwise, operators are duplicated.

3.3.1 Bounding Search Space with the Top-Down Algorithm

Recall our definition of β in Section 3.2.1. We now show in Theorem 4 that β also represents the the upper bound on the ratio of the search space of the Top-Down algorithm to that of the exhaustive search. Note that, as the ratio $\frac{max_{cs}}{N}$ decreases linearly, β decreases exponentially. When $max_{cs} << N$, β is orders of magnitude less than 1. Thus, the search space of the Top-Down algorithm is orders of magnitude less than the exhaustive search space.

THEOREM 4. *If $\mathcal{O}_{top-down}$ represents the solution space for the top-down algorithm, then*

$$\mathcal{O}_{top-down} \leq \beta \mathcal{O}_{exhaustive}$$

PROOF. The worst case search space of the Top-Down algorithm results when all query tree nodes (sources, operators and sink) appear in the same cluster. As in the case of Theorem 2 we compute this search space by considering all possible query trees and all possible placements of operators within a single cluster at each level.

We are given a network with N nodes, and a query $\mathcal{Q}$ over K streams $\langle S_1, S_2, \ldots S_K \rangle$. At the top level t, we have K streams. The search space $\mathcal{O}_t$ at level t follows from Equation 2 and is given by:

$$\mathcal{O}_t = \left(\frac{K! \times (K-1)!}{2^{K-1}} \right) \times (max_{cs})^{(K-1)}$$

In the worst case the coordinator at each level may assign all streams to a single partition thereby causing the search space to be the same at all levels. Thus, $\mathcal{O}_{top-down}$ is given by

$$\mathcal{O}_{top\text{-}down} \leq h \times \left(\frac{K! \times (K-1)!}{2^{K-1}} \right) \times (max_{cs})^{(K-1)} \quad (5)$$

Thusfrom Equation 5 and Lemma 1 we have

$$\mathcal{O}_{top-down} \leq \beta \mathcal{O}_{exhaustive}$$

□

3.3.2 Sub-Optimality in the Top-Down Algorithm

The Top-Down algorithm works by propagating a query down the network hierarchy, described in Section 3.1.1. Assume that we are given a query $\mathcal{Q}$ and a network N organized as a hierarchy. At each level, the coordinator chooses a deployment $\mathcal{M}$ (and hence a corresponding tree $\mathcal{P}$) with the least cost for the sub-query assigned to it, and then assigns the operators in this tree to nodes in its underlying cluster. It follows from Theorem 1 that the maximum network approximation is incurred at the top most level of the network hierarchy and therefore the Top-Down algorithm is most sub-optimal when all the edges of tree $\mathcal{P}$ are deployed at the top-most level. The following theorem establishes the bounds on sub-optimality of the top-down algorithm as compared to an optimal deployment using an exhaustive search.

THEOREM 5. *A query $\mathcal{Q}$ deployed using the Top-Down algorithm over a network N is no more than*

$$\sum_{e_k \in E^Q} \left(\sum_{i=1}^{i=h-1} 2d_i \right) \times s_k$$

sub-optimal compared to the optimal deployment of query $\mathcal{Q}$ over the same network N, where h is the number of levels in the network hierarchy of N, E^Q represents the set of edges of the tree chosen for query $\mathcal{Q}$, d_i is the maximum intra-cluster traversal cost at level i and s_k is the stream rate for the k^{th} edge e_k.

PROOF. The maximum sub-optimality of the Top-Down algorithm occurs only when all the edges of the tree chosen for $\mathcal{Q}$ are mapped to the top-most level, i.e. no two nodes (operators or sources or sinks) lie in the same underlying cluster. The proof then follows directly from Theorem 1. □

3.4 Bottom-Up versus Top-Down

The Bottom-Up and Top-Down algorithms bound the search space by limiting exhaustive searches to a bounded cluster. However, the Bottom-Up algorithm considers only a subset of operators at each level thus pruning the search space even more aggressively. As shown in Theorem 2 and 4, in the worst case both algorithms may result in the same search space. However, as we show in our experiments, in the average case the Bottom-Up algorithm must consider only about 50% of the plans of the Top-Down algorithm.

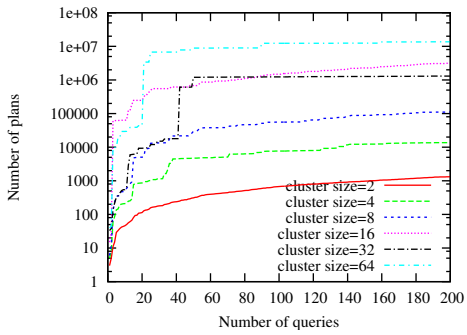

Figure 8: Bottom-Up Algorithm: Number of plans

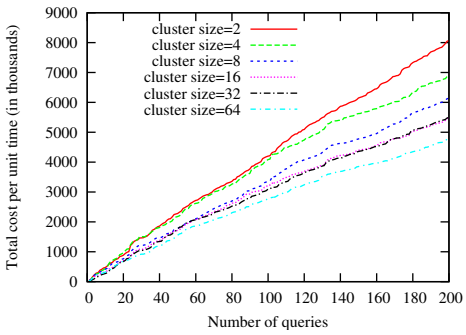

Figure 9: Bottom-Up Algorithm: Cost

Since the Bottom-Up algorithm prunes the search space aggressively, it does not consider all join orders. In particular, join orders where remote streams are joined before locally available streams are not considered. As a result in the worst case, the Bottom-Up algorithm may choose plans that are arbitrarily bad. However, it is possible to identify these cases a priori by considering stream data rates and join selectivities. In cases where the variation in the cost of different join orderings of the same query is high, it may be preferable to use the Top-Down algorithm, which offers tighter bounds on the sub-optimality of the deployment. However, if the network placement costs are the dominant factor in deployment costs, or if deployment time is a primary concern as compared to sub-optimality (which may be true in the case of short-lived queries), the Bottom-Up algorithm is ideal. The Bottom-Up algorithm offers smaller time-to-deploy for queries since the algorithm completes at the lowest level where all sources are found. On the other hand, with Top-Down the query always traverses the entire depth of the hierarchy.

It is possible to combine the two approaches into a hybrid approach. For example, we could reduce both sub-optimality and deployment time while considering a larger search space by beginning the Bottom-Up algorithm from some level $j > 1$ higher in the clustering hierarchy. We leave such approaches to future work.

4. EXPERIMENTS

We present both simulation based experiments and experiments conducted on Emulab [1] using IFLOW [22], our implementation of a distributed data stream system that supports distributed deployment of continual-queries. In particular, we show in our experiments that in the average case the Top-Down algorithm is only 10% sub-optimal compared to an exhaustive search, while the Bottom-Up algorithm is 34% sub-optimal. However, the deployment time of the Bottom-Up algorithm is 70% less than that of the Top-Down algorithm.

4.1 Experimental Setup

Our simulation experiments were conducted over transit-stub topology networks generated using the standard tool, the GT-ITM internetwork topology generator [35]. Most experiments were conducted using a 128 node network, with a standard Internet-style topology: 1 transit (e.g. "backbone") domain of 4 nodes, and 4 "stub" domains connected to the transit domain (each of 8 nodes). Link costs (per byte transferred) were assigned such that the links in the stub domains had lower costs than those in the transit domain, corresponding to transmission within an intranet being far cheaper than long-haul links. The *cost of a deployment* is the total data transferred along each link times the link cost.

We used a synthetic workload so that we could experiment with a large variety of stream rates, query complexities, and operator selectivities. Our workload was generated using a uniformly random workload generator. The workload generator generated stream rates, selectivities and source placements for a specified number of streams according to a uniform distribution. It also generated queries with the number of joins per query varying within a specified range (2-5 joins per query) with random sink placements. In our experiments we use a cost formulation that tries to minimize the communication cost incurred per unit time by the deployed query plan. Therefore, as described in Section 3.1.1 our network is organized into a clustering hierarchy based on link costs which represent the cost of transmitting a unit amount of data across the link.

4.2 Tuning Cluster Size: Tradeoff between Sub-Optimality and Search Space

As explained in Section 3, an exhaustive search of all possible query plans and all possible placement of operators may not be feasible as network size increases. For example, an exhaustive search on a 128 node network for the deployment of a single query over 5 stream sources required enumeration of approximately 4.83×10^{10} plans that took nearly 3 hours to complete on our system.

In this section we demonstrate how the max_{cs} parameter can be used to tune the tradeoff between the sub-optimality of the heuristic and minimizing the search space. Intuitively, a larger max_{cs} means: (1) larger clusters and (2) fewer levels in the hierarchy. Moreover, larger clusters increase the chance that multiple sources will be found in the same cluster. The impact of these effects are studied below.

The experiments were conducted on the 128 node topology described in Section 4.1, with 10 source streams. We averaged our results over 10 workloads generated using our random workload generator, each with a different set of placements for sources and sinks. Each workload consisted of 200 queries with 2-5 joins per query.

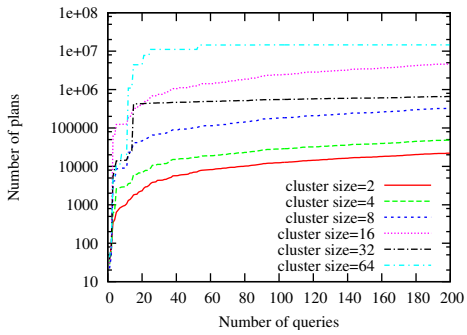

Figure 10: Top-Down Algorithm: Number of plans

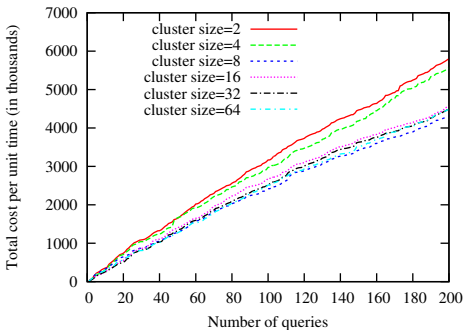

Figure 11: Cost using Top-Down algorithm

4.2.1 Bottom-Up Algorithm: Effect of Cluster Size on Search Space

In this experiment we studied the effect of the cluster size parameter max_{cs} on the search space with the Bottom-Up algorithm. Figure 8 depicts the cumulative number of plans examined on a log scale, with varying max_{cs}. As the figure shows, the number of plans increases as max_{cs} increases. A max_{cs} value of 2 resulted in the fewest plans examined. This is because the resulting hierarchy had small cluster sizes at each level, and thus fewer deployment options had to be examined.

Interestingly, we notice that a max_{cs} value of 32 results in a smaller search space than a value of 16. In both cases, the hierarchy had the same number of levels, but in the case of $max_{cs} = 32$, the upper level clusters were smaller (since the lower level clusters were larger.) Since many sources are found remotely, most of the planning is done at the upper levels, and having small cluster sizes at those levels results in a smaller search space overall.

In contrast, a max_{cs} value of 64 resulted in the maximum search space, nearly an order of magnitude larger than $max_{cs} = 16$. This is a straightforward effect of the increased probability of finding sources in a larger cluster. For example a query over 4 streams, with all streams found within a 57 node *Level 1* cluster, considers as many as 3.3×10^6 deployments.

4.2.2 Bottom-Up Algorithm: Effect of Cluster Size on Cost

The aim of this experiment was to study the impact of the max_{cs} parameter on the cost of deployments obtained with the Bottom-Up algorithm. Figure 9 shows the cumulative deployed cost per unit time of queries deployed incrementally for different values of the max_{cs} parameter. It can be noticed that cost decreases as the max_{cs} value is increased. For example, a max_{cs} value of 64 results in a 21% decrease in cost compared to a max_{cs} value of 8. With smaller cluster sizes, the number of levels in the hierarchy increases. As a result, more deployments are computed at higher levels resulting in greater approximations.

To summarize, in terms of sub-optimality, fewer levels and more nodes per level is best. In terms of search space, fewer nodes per level is best. A useful guideline for choosing max_{cs} for the Bottom-Up algorithm is:

- Choose the largest value of max_{cs} that results in a search space (using Theorem 2) that is acceptable.

4.2.3 Top-Down Algorithm: Effect of Cluster Size on Search Space

Figure 10 shows the effect of the max_{cs} parameter on the search space of the Top-Down algorithm. Note that the y-axis has a log scale. As the figure shows, $max_{cs} = 2$ results in the smallest search space, because the top level cluster must have at most 2 nodes. We see (as before) that $max_{cs} = 32$ is better than $max_{cs} = 16$. Setting $max_{cs} = 32$ results in fewer nodes at the top level, where the whole query is considered for deployment. Although $max_{cs} = 64$ also results in a small top level cluster, in this case individual lower level clusters have much higher probability of having multiple sources. The result is that usually the whole query must be passed down to a large lower level cluster, so that many operators and deployments must be considered. In general smaller values of max_{cs} results in smaller search spaces.

4.2.4 Top-Down Algorithm: Effect of Cluster Size on Cost

Next, Figure 11 shows the effect of the cluster size parameter max_{cs} on the cost in the Top-Down algorithm. Note that large values of max_{cs} (> 4) result in deployed costs that are close to each other. The Top-Down algorithm considers all possible operator orderings at the top-most level (regardless of max_{cs}). This in turn results in a good and mostly 'similar' choice of operator ordering for a range of max_{cs} values. However, if max_{cs} is too small, there are many levels in the hierarchy. Since each level adds more inaccuracy to the approximation, each top-level node provides a poorer picture of the underlying network. Therefore, the Top-Down algorithm makes poorer planning decisions.

To limit sub-optimality, we need a reasonably large max_{cs} (to avoid the above effect.) To bound search space, we need a small max_{cs}. Therefore, a useful guideline for choosing max_{cs} for the Top-Down algorithm is:

- Choose the smallest value of max_{cs} that is large enough so that the height of the hierarchy results in reasonable sub-optimality (based on Theorem 5).

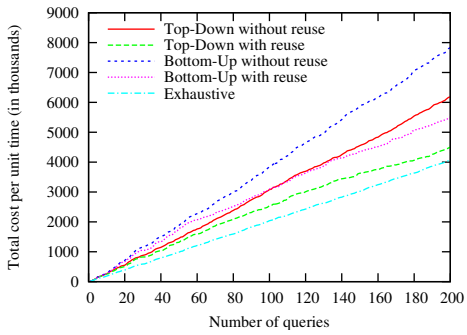

Figure 12: Effect of reuse and comparison with exhaustive search

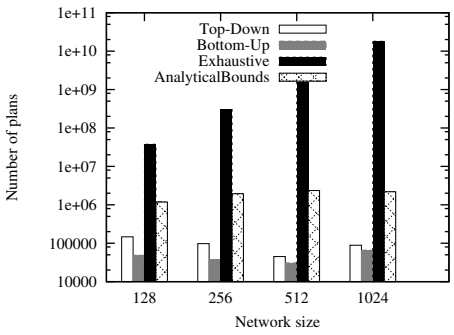

Figure 13: Average number of plans

4.3 Effect of Reuse and Comparison with Exhaustive Search

We now examine the effect of operator reuse in the Bottom-Up and Top-Down algorithms. Figure 12 shows the cumulative cost of deployment using exhaustive search and the two algorithms both with and without operator reuse. The max_{cs} parameter was set to 32. We chose this value of max_{cs} based on the above guideline for the Bottom-Up algorithm; and we used the same value for the Top-Down to provide apples-to-apples comparison. Operator reuse was implemented through stream-advertisements. The communication cost of advertisements was negligible compared to the data streams themselves. The figure shows that the Bottom-Up algorithm benefits by nearly a 30% decrease in cost per unit time through operator reuse, while the Top-Down algorithm achieves cost saving of 27% per unit time through operator reuse.

Figure 12 also allows us to compare the deployed costs of the two algorithms with an exhaustive search. As can be seen, the Top-Down algorithm with reuse performs nearly 19% better than the Bottom-Up algorithm with reuse. This is because the Bottom-Up algorithm may choose a sub-optimal plan since it does not consider an ordering of all operators at any level, unlike the Top-Down algorithm. When compared to an exhaustive search, the Bottom-Up algorithm with reuse, performs sub-optimally by only 34% and the Top-Down algorithm by only 10%. This shows that the sub-optimality due to the approximations made by the algorithms is minimal. We show in our next experiment that both algorithms effect a massive decrease in search space.

4.4 Scalability with Network Size

In this experiment we study the scalability of the algorithms with respect to the number of deployments considered as network size increases. We generated a workload of 100 queries using 10 stream sources with each query performing joins over 4 streams. We measured the average number of deployments considered for queries in this workload over 4 different transit-stub topologies of different sizes generated using GT-ITM. Again, sinks were placed at random nodes in the network. Figure 13 shows the deployments considered for a single query with Bottom-Up and Top-Down algorithms with max_{cs} 32 and exhaustive search. The figure also shows how the average case (experimental) compares with the worst case (theoretical) analytical bounds. Note that the y-axis has a log scale. Again, the value of max_{cs} was set to 32 to produce the largest feasible search space. Note that the increase in $\mathcal{O}_{exhaustive}$ is offset by the decrease in β such that the worst case bounds are nearly identical across the different networks.

The values for exhaustive search were calculated using Lemma 1 while the analytical bounds was calculated using Theorems 2 and 4. Clearly, performing exhaustive searches in such systems is infeasible. Both the Top-Down and Bottom-Up algorithms decrease the search space by at least 99%. We also see that the search space per query in the Bottom-Up algorithm is nearly 45% less than that of the Top-Down algorithm. This can be attributed to the early splitting of queries between levels in the Bottom-Up algorithm resulting in fewer operators being considered for placement at each level. Meanwhile, the Top-Down algorithm must consider all operator deployments at all levels in the hierarchy.

Although the search space of Top-Down and Bottom-Up algorithms seems to first decrease with network size and then increase, note that this is only a particular characteristic of our sample networks. For example, clustering using max_{cs} 32 resulted in an average *Level 1* cluster size of 26 with a 128-node network, and 15 with a 510-node network. Thus the search space for a 510-node network is less than that of the 128 node network. Note that the search space, while being limited by the max_{cs} parameter, is affected by the average cluster size too, which depends on the particular network topology.

4.5 Prototype Experiments

The next set of experiments was conducted on Emulab using IFLOW [22], our implementation of the distributed data stream system which supports hierarchies and advertisements as described earlier. The testbed on Emulab consisted of 32 nodes (Intel XEON, 2.8 GHz, 512MB RAM, RedHat Linux 9), organized into a topology that was again generated with GT-ITM. Links were 100Mbps and the inter-node delays were set between 1msec and 6msec. The query workload for the following experiments consisted of 25 queries over 8 stream sources. The number of streams per query varied from 2 to 4.

4.5.1 Deployment Time

The first experiment conducted on Emulab was meant to validate our claim about the stricter search space bounds

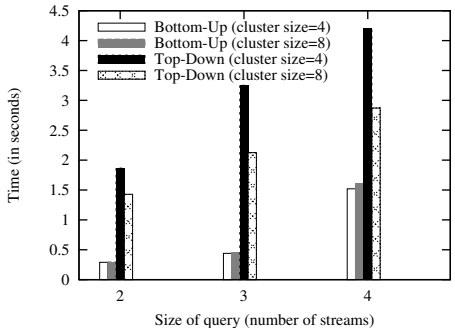

Figure 14: Query deployment time

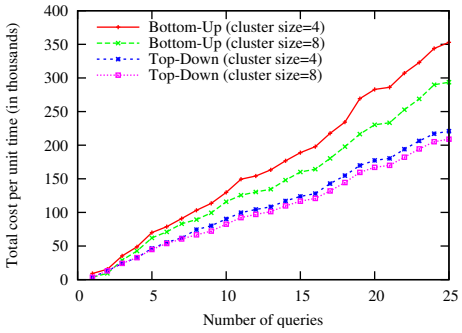

Figure 15: Utility with Bottom-Up and Top-Down algorithms

offered by the Bottom-Up algorithm. Figure 14 shows the average deployment time in seconds for different query sizes. We observe that the deployment times of the Bottom-Up algorithm is almost 70% less than that of the Top-Down algorithm. This can be attributed to two factors: (1) the smaller search space in the Bottom-Up algorithm, and (2) the fact that the Top-Down algorithm must always traverse the entire depth of the network hierarchy. We also observe that the deployment time of the Top-Down algorithm decreases with increasing max_{cs} value. This is due to the fact that more levels need to be traversed for lower max_{cs} values, hence resulting in higher deployment times.

4.5.2 Deployment Cost

In this experiment, we studied the cost of deployments with the Bottom-Up and Top-Down algorithms for different values of max_{cs}. Figure 15 shows the cumulative cost incurred per unit time over 25 queries. We observe that the Top-Down algorithm offers a lower deployed cost than the Bottom-Up algorithm. This is in alignment with our simulation results. Since the Top-Down algorithm considers all operator orderings at the top-most level this algorithm leads to the selection of a better execution plan.

5. RELATED WORK

Distributed query optimization has received a great deal of attention from researchers since the 1980s [21]. Classic efforts in this area include R* [33], Distributed INGRES [31] and SDD-1 [9]. Both the R* algorithm and the Distributed INGRES algorithm execute at a *master* site. Since our system may consist of thousands of nodes, it is infeasible to maintain all network information at a single node or perform exhaustive searches for an optimal deployment. Note that our notion of local and global views is with reference to the actual network locations of sources and cluster boundaries. This is very different from the LAV, GAV and GLAV [18] concepts used for schema mapping in data-integration systems. Also, our algorithms primarily deal with performance issues and require orthogonal research in semantic issues such as schema mapping and transformation [27].

A number of data-stream systems such as STREAM [5], Borealis [2], TelegraphCQ [10] and NiagaraCQ [13] have been developed to process queries over continuous streams of data. Approaches to query optimization in centralized stream processing systems have explored use of techniques like common sub-expression elimination and commutative ordering of operators [7, 12]. There has also been an effort in moving towards distributed processing of stream queries [2, 29] in general and query optimization [4, 32] in these systems in particular. Novel systems like Eddies [16, 4] have also used a tuple-by-tuple routing approach to adaptively decide the execution plan of a query. The paradigm of in-network query processing has been used earlier in sensor networks [23, 34]. The use of this technique in stream based systems to decide operator placement when the query tree is already known is described in [3] and a heuristic based approach for the same problem is presented in [26]. Optimal placement for a single query on a sensor network is considered in [30]. However, we consider the more generic problem of selecting an optimal plan and do so by simultaneously considering operator ordering and placements.

Although our algorithms utilize stream rates and statistics we do not deal with the issue of computing and maintaining these statistics. Rate-based and sliding-window approaches have been described elsewhere for computing statistics in data stream systems [6, 32]. The problems of common sub-expression elimination and operator reuse in our distributed stream processing system, overlaps to some extent with the problems of view selection, maintenance and reuse [20, 11, 19]. However, our solution space is compounded by the need to additionally consider network costs. The problem of multi-query optimization in distributed data stream systems was defined in [28], but that work only sketched out preliminary solution approaches. We build upon this work by presenting detailed algorithms for query optimization, and conducting an in-depth experimental study of their effectiveness. Note that our algorithms can be applied to the task scheduling problem [15] in distributed systems, but are designed to deal with distribution at a much larger scale.

6. CONCLUSION & FUTURE WORK

We described the query-optimization problem in distributed data-stream systems and illustrated how traditional database paradigms may not apply to such systems. We demonstrated through our experiments that selection of an optimal execution plan in such systems must consider operator ordering, network placement and operator reuse. We presented a query optimization infrastructure that has two key components: a hierarchical clustering of network nodes that allow network approximations and stream advertisements that enable operator reuse. We described algorithms *Top-Down* and *Bottom-Up* that find efficient execution plans

while examining a very small search space. Experimental and analytical results showed that both algorithms offer costs that are comparable to that provided by an exhaustive search while exploring much fewer plans. In ongoing work we are exploring run-time reconfiguration of deployments, and other optimization opportunities achievable through query containment.

7. REFERENCES

[1] Emulab - network emulation testbed. *http://www.emulab.net/*.

[2] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[3] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In *VLDB*, 2004.

[4] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[6] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *PODS*, 2003.

[7] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, 2004.

[8] J. Beaver and M. A. Sharaf. Location-aware routing for data aggregation for sensor networks. In *Geo Sensor Networks Workshop*, 2003.

[9] P. A. Bernstein et al. Query processing in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 1981.

[10] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[11] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, 1995.

[12] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, 2002.

[13] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[14] R. Chirkova and C. Li. Materializing views with minimal size to answer queries. In *PODS*, 2003.

[15] R. Chow and T. Johnson. *Distributed Operating Systems and Algorithms*. Addison-Wesley Longman, Reading, MA, 1997.

[16] A. Deshpande and J. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.

[17] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *SIGMOD*, 1978.

[18] M. Friedman, A. Levy, and T. Millstein. Navigational plans for data integration. In *AAAI*, 1999.

[19] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *SIGMOD*, 2001.

[20] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 2001.

[21] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 2000.

[22] V. Kumar et al. Implementing diverse messaging models with self-managing properties using IFLOW. In *IEEE International Conference on Autonomic Computing*, 2006.

[23] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[24] J. Moy. OSPF version 2, request for comments 2328. 1998.

[25] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, 2003.

[26] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[27] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 2001.

[28] S. Seshadri, V. Kumar, and B. F. Cooper. Optimizing multiple queries in distributed data stream systems. In *2nd Workshop on Networking Meets Database (NetDB), in conjunction with ICDE*, 2006.

[29] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Technical Report CS-02-1205, U.C. Berkeley*, 2002.

[30] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *PODS*, 2005.

[31] M. Stonebraker. The design and implementation of distributed INGRES. *The INGRES papers: anatomy of a relational database system*, 1986.

[32] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, 2002.

[33] R. Williams et al. *R*: An overview of the architecture.*

[34] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 2002.

[35] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Infocom*, 1996.